The Memory Hierarchy
Caches, and VM
CS 740

10/2/2013

Topics
• The memory hierarchy
• Cache design
• Virtual Memory
Ideal Memory

Zero access time (latency)
Infinite capacity
Zero cost
Infinite bandwidth (to support multiple accesses in parallel)
The Problem

Ideal memory's requirements oppose each other

Bigger is slower
  • Bigger → Takes longer to determine the location

Faster is more expensive
  • Memory technology: SRAM vs. DRAM

Higher bandwidth is more expensive
  • Need more banks, more ports, higher frequency, or faster technology
Computer System
The Tradeoff

(size: 608 B, speed: 1.4 ns, $/Mbyte: $90/MB, block size: 4 B)

(size: 128k B, speed: 4.2 ns, $/Mbyte: $90/MB, block size: 4 B)

(size: 512kB -- 4MB, speed: 16.8 ns, $/Mbyte: $2-6/MB, block size: 16 B)

(size: 128 MB, speed: 112 ns, $/Mbyte: $2-6/MB, block size: 4-8 KB)

(size: 27GB, speed: 9 ms, $/Mbyte: $0.01/MB, block size: 4-8 KB)

larger, slower, cheaper

(Numbers are for a 21264 at 700MHz circa ~2000)
Why is bigger slower?

- Physics slows us down
- Racing the speed of light: \((3.0 \times 10^8 \text{ m/s})\)
  - clock = 500MHz
  - how far can I go in a clock cycle?
  - \((3.0 \times 10^8 \text{ m/s}) / (500 \times 10^6 \text{ cycles/s}) = 0.6 \text{ m/cycle}\)
  - For comparison: 21264 is about 17mm across
- Capacitance:
  - long wires have more capacitance
  - either more powerful (bigger) transistors required, or slower
  - signal propagation speed proportional to capacitance
  - going “off chip” has an order of magnitude more capacitance
Caches:
L1 data
L1 instruction
L2 unified
+ L3 off-chip
Caches:
L1 data
L1 instruction
L2 unified
+ L3 off-chip
Memory in a Modern System
Locality of Reference

Principle of Locality:

- Programs tend to reuse data and instructions near those they have used recently.
- **Temporal locality**: recently referenced items are likely to be referenced in the near future.
- **Spatial locality**: items with nearby addresses tend to be referenced close together in time.

Locality in Example:

- **Data**
  - Reference array elements in succession (spatial)
- **Instructions**
  - Reference instructions in sequence (spatial)
  - Cycle through loop repeatedly (temporal)

```c
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
*v = sum;
```
Caching: The Basic Idea

Main Memory
- Stores words
  A-Z in example

Cache
- Stores subset of the words
  4 in example
- Organized in lines
  - Multiple words
  - To exploit spatial locality

Access
- Word must be in cache for processor to access
How important are caches?

- 21264 Floorplan
- Register files in middle of execution units
- 64k instr cache
- 64k data cache
- Caches take up a large fraction of the die

(Figure from Jim Keller, Compaq Corp.)
Between any two levels, memory is divided into lines (aka “blocks”).
Data moves between levels on demand, in line-sized chunks.
Invisible to application programmer
- Hardware responsible for cache operation
Upper-level lines a subset of lower-level lines

Access word w in line a (hit)
Access word v in line b (miss)
A Modern Memory Hierarchy

Register File
32 words, sub-nsec

L1 cache
~32 KB, ~nsec

L2 cache
512 KB ~ 1MB, many nsec

L3 cache,
.....

Main memory (DRAM),
GB, ~100 nsec

Swap Disk
100 GB, ~10 msec

Memory Abstraction

Manual/Compiler
Register Spilling

Automatic
HW Cache
Management

Automatic
Demand
Paging
Key Questions:

- Where should a line be placed in the cache? (line placement)
- How is a line found in the cache? (line identification)
- Which line should be replaced on a miss? (line replacement)
- What happens on a write? (write strategy)

Constraints:

- Design must be very simple
  - Hardware realization
  - All decision making within nanosecond time scale
- Want to optimize performance for “typical” programs
  - Do extensive benchmarking and simulations
  - Many subtle engineering tradeoffs
Direct-Mapped Caches

Simplest Design
- Each memory line has a unique cache location

Parameters
- Line (aka block) size $B = 2^b$
  - Number of bytes in each line
  - Typically 2X-8X word size
- Number of Sets $S = 2^s$
  - Number of lines cache can hold
- Total Cache Size $= B*S = 2^{b+s}$

Physical Address
- Address used to reference main memory
- $n$ bits to reference $N = 2^n$ total bytes
- Partition into fields
  - Offset: Lower $b$ bits indicate which byte within line
  - Set: Next $s$ bits indicate how to locate line within cache
  - Tag: Identifies this line when in cache
Indexing into Direct-Mapped Cache

- Use set index bits to select cache set

Set 0:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>B−1</td>
<td></td>
</tr>
</tbody>
</table>

Set 1:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>B−1</td>
<td></td>
</tr>
</tbody>
</table>

Set S−1:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>B−1</td>
<td></td>
</tr>
</tbody>
</table>

Physical Address

tag  set index  offset
Direct-Mapped Cache Tag Matching

Identifying Line

- Must have tag match high order bits of address
- Must have $\text{Valid} = 1$

*Lower bits of address select byte or word within cache line*

Physical Address

- $t$ (tag)
- $s$ (set index)
- $b$ (offset)
Properties of Direct Mapped Caches

**Strength**
- Minimal control hardware overhead
- Simple design
- (Relatively) easy to make fast

**Weakness**
- Vulnerable to thrashing
- Two heavily used lines have same cache index
- Repeatedly evict one to make room for other
Vector Product Example

```c
float dot_prod(float x[1024], y[1024])
{
    float sum = 0.0;
    int i;
    for (i = 0; i < 1024; i++)
        sum += x[i]*y[i];
    return sum;
}
```

Machine
- DECStation 5000
- MIPS Processor with 64KB direct-mapped cache, 16 B line size

Performance
- Good case: 24 cycles / element
- Bad case: 66 cycles / element
Thrashing Example

- Access one element from each array per iteration
Thrashing Example: Good Case

Access Sequence
- Read x[0]
  - x[0], x[1], x[2], x[3] loaded
- Read y[0]
  - y[0], y[1], y[2], y[3] loaded
- Read x[1]
  - Hit
- Read y[1]
  - Hit
- ... ...
- 2 misses / 8 reads

Analysis
- x[i] and y[i] map to different cache lines
- Miss rate = 25%
  - Two memory accesses / iteration
  - On every 4th iteration have two misses

Timing
- 10 cycle loop time
- 28 cycles / cache miss
- Average time / iteration = 10 + 0.25 * 2 * 28
Thrashing Example: Bad Case

Access Pattern
- Read \( x[0] \)
  - \( x[0], x[1], x[2], x[3] \) loaded
- Read \( y[0] \)
  - \( y[0], y[1], y[2], y[3] \) loaded
- Read \( x[1] \)
  - \( x[0], x[1], x[2], x[3] \) loaded
- Read \( y[1] \)
  - \( y[0], y[1], y[2], y[3] \) loaded
  - \( \ldots \)
- 8 misses / 8 reads

Analysis
- \( x[i] \) and \( y[i] \) map to same cache lines
- Miss rate = 100%
  - Two memory accesses / iteration
  - On every iteration have two misses

Timing
- 10 cycle loop time
- 28 cycles / cache miss
- Average time / iteration = 10 + 1.0 * 2 * 28
Miss Types

Compulsory Misses – required to warm up the cache

Capacity Misses – occur when the cache is full

Conflict Misses – Block placement may cause these in direct or non-fully associative caches

Coherence Misses – occur because of invalidations caused between threads
Set Associative Cache

Mapping of Memory Lines

- Each set can hold E lines (usually E=2–8)
- Given memory line can map to any entry within its given set

Eviction Policy

- Which line gets kicked out when bring new line in
- Commonly either “Least Recently Used” (LRU) or pseudo-random
  - LRU: least-recently accessed (read or written) line gets evicted

```
Set i:

Line 0:  Tag  Valid
        0    1    •••  B–1

Line 1:  Tag  Valid
        0    1    •••  B–1

        ...

Line E–1:  Tag  Valid
         0    1    •••  B–1
```

LRU State
Indexing into 2-Way Associative Cache

- Use middle $s$ bits to select from among $S = 2^s$ sets

<table>
<thead>
<tr>
<th>Set 0:</th>
<th></th>
<th>Set 1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Valid</td>
<td>0</td>
</tr>
<tr>
<td>Tag</td>
<td>Valid</td>
<td>0</td>
</tr>
<tr>
<td>Tag</td>
<td>Valid</td>
<td>0</td>
</tr>
<tr>
<td>Tag</td>
<td>Valid</td>
<td>0</td>
</tr>
</tbody>
</table>

Physical Address

- tag
- set index
- offset
Associative Cache Tag Matching

Identifying Line

- Must have one of the tags match high order bits of address
- Must have Valid = 1 for this line

Lower bits of address select byte or word within cache line

Tag | Valid
--- | ---
0   | 1   • • • B–1
0   | 1   • • • B–1

Physical Address: tag | set index | offset
Two-Way Set Associative Cache Implementation

- Set index selects a set from the cache
- The two tags in the set are compared in parallel
- Data is selected based on the tag result
Fully Associative Cache

Mapping of Memory Lines

- Cache consists of single set holding E lines
- Given memory line can map to any line in set
- Only practical for small caches

Entire Cache

<table>
<thead>
<tr>
<th>Line 0:</th>
<th>Tag</th>
<th>Valid</th>
<th>0</th>
<th>1</th>
<th>•••</th>
<th>B–1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 1:</td>
<td>Tag</td>
<td>Valid</td>
<td>0</td>
<td>1</td>
<td>•••</td>
<td>B–1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line E–1:</td>
<td>Tag</td>
<td>Valid</td>
<td>0</td>
<td>1</td>
<td>•••</td>
<td>B–1</td>
</tr>
</tbody>
</table>
Fully Associative Cache Tag Matching

Identifying Line

- Must check all of the tags for match
- Must have \( \text{Valid} = 1 \) for this line

Lower bits of address select byte or word within cache line
Replacement Algorithms

• When a block is fetched, which block in the target set should be replaced?

Optimal algorithm:
  - replace the block that will not be used for the longest period of time
  - must know the future

Usage based algorithms:
  • Least recently used (LRU)
    - replace the block that has been referenced least recently
    - hard to implement (unless low associativity)
  • Not most recently used
  • Victim/next-victim

Non-usage based algorithms:
  • First-in First-out (FIFO)
    - treat the set as a circular queue, replace block at head of queue.
    - Essentially replace oldest
  • Random (RAND)
    - replace a random block in the set
    - even easier to implement
Implementing RAND and FIFO

FIFO:
- maintain a modulo E counter for each set.
- counter in each set points to next block for replacement.
- increment counter with each replacement.

RAND:
- maintain a single modulo E counter.
- counter points to next block for replacement in any set.
- increment counter according to some schedule:
  - each clock cycle,
  - each memory reference, or
  - each replacement anywhere in the cache.

LRU
- Need state machine for each set
- Encodes usage ordering of each element in set
- E! possibilities ==> ~ E log E bits of state
Write Policy

• What happens when processor writes to the cache?
• Should memory be updated as well?

Write Through:
• Store by processor updates cache and memory
• Memory always consistent with cache
• Never need to store from cache to memory
• ~2X more loads than stores
Write Policy (Cont.)

Write Back:

- Store by processor only updates cache line
- Modified line written to memory only when it is evicted
  - Requires “dirty bit” for each line
    » Set when line in cache is modified
    » Indicates that line in memory is stale
- Memory not always consistent with cache
Write Buffering

Write Buffer
- Common optimization for all caches
- Overlaps memory updates with processor execution
- Read operation must check write buffer for matching address
Multi-Level Caches

Options: *separate* data and instruction caches, or a *unified* cache

How does this affect self modifying code?
Bandwidth Matching

Challenge
- CPU works with short cycle times
- DRAM (relatively) long cycle times
- How can we provide enough bandwidth between processor & memory?

Effect of Caching
- Caching greatly reduces amount of traffic to main memory
- But, sometimes need to move large amounts of data from memory into cache

Trends
- Need for high bandwidth much greater for multimedia applications
  - Repeated operations on image data
- Recent generation machines (e.g., Pentium II) greatly improve on predecessors
High Bandwidth Memory Systems

**Solution 1**
High BW DRAM

Example:
- Page Mode DRAM
- RAMbus

**Solution 2**
Wide path between memory & cache

Example: Alpha AXP 21064
- 256 bit wide bus, L2 cache, and memory.
Cache Performance Metrics

Miss Rate
- fraction of memory references not found in cache (misses/references)
- Typical numbers:
  - 3-10% for L1
  - can be quite small (e.g., < 1%) for L2, depending on size, etc.

Hit Time
- time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)
- Typical numbers:
  - 1-3 clock cycles for L1
  - 3-12 clock cycles for L2

Miss Penalty
- additional time required because of a miss
  - Typically 25-100 cycles for main memory
Hierarchical Latency Analysis

For a given memory hierarchy level \( i \) it has a technology-intrinsic access time of \( t_i \). The perceived access time \( T_i \) is longer than \( t_i \).

Except for the outer-most hierarchy, when looking for a given address there is

- a chance (hit-rate \( h_i \)) you “hit” and access time is \( t_i \)
- a chance (miss-rate \( m_i \)) you “miss” and access time \( t_i + T_{i+1} \)
- \( h_i + m_i = 1 \)

Thus

\[
T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1})
\]
\[
T_i = t_i + m_i \cdot T_{i+1}
\]

keep in mind, \( h_i \) and \( m_i \) are defined to be the hit-rate and miss-rate of just the references that missed at \( L_{i-1} \)
Hierarchy Design Considerations

Recursive latency equation

\[ T_i = t_i + m_i \cdot T_{i+1} \]

The goal: achieve desired \( T_1 \) within allowed cost

\( T_i \approx t_i \) is desirable

Keep \( m_i \) low

- increasing capacity \( C_i \) lowers \( m_i \), but beware of increasing \( t_i \)
- lower \( m_i \) by smarter management (replacement::anticipate what you don’t need, prefetching::anticipate what you will need)

Keep \( T_{i+1} \) low

- faster lower hierarchies, but beware of increasing cost
- introduce intermediate hierarchies as a compromise
## Intel Pentium 4 Example

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>90nm P4, 3.6 GHz</strong></td>
<td></td>
</tr>
<tr>
<td><strong>L1 D-cache</strong></td>
<td>• $C_1 = 16K$</td>
</tr>
<tr>
<td></td>
<td>• $t_1 = 4$ cycle int / 9 cycle fp</td>
</tr>
<tr>
<td><strong>L2 D-cache</strong></td>
<td>• $C_2 = 1024$ KB</td>
</tr>
<tr>
<td></td>
<td>• $t_2 = 18$ cycle int / 18 cycle fp</td>
</tr>
<tr>
<td><strong>Main memory</strong></td>
<td>• $t_3 = \sim 50$ ns or 180 cycle</td>
</tr>
<tr>
<td><strong>Notice</strong></td>
<td>• best case latency is not 1</td>
</tr>
<tr>
<td></td>
<td>• worst case access latencies are into 500+ cycles</td>
</tr>
</tbody>
</table>

### Timing Values

- If $m_1=0.1$, $m_2=0.1$
  - $T_1=7.6$, $T_2=36$
- If $m_1=0.01$, $m_2=0.01$
  - $T_1=4.2$, $T_2=19.8$
- If $m_1=0.05$, $m_2=0.01$
  - $T_1=5.00$, $T_2=19.8$
- If $m_1=0.01$, $m_2=0.50$
  - $T_1=5.08$, $T_2=108$
Impact of Cache and Block Size

**Cache Size**
- Effect on miss rate?
- Effect on hit time?

**Block Size**
- Effect on miss rate?
- Effect on miss penalty?
- Effect on hit time?
Impact of Associativity

- Direct-mapped, set associative, or fully associative?
- Total Cache Size (tags+data)?
- Miss rate?
- Hit time?
- Miss Penalty?
# Impact of Replacement Strategy

- RAND, FIFO, or LRU?
- Total Cache Size (tags+data)?
- Miss Rate?
- Miss Penalty?
<table>
<thead>
<tr>
<th>Impact of Write Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write-through or write-back?</td>
</tr>
<tr>
<td><strong>Advantages of Write Through?</strong></td>
</tr>
<tr>
<td><strong>Advantages of Write Back?</strong></td>
</tr>
</tbody>
</table>
Allocation Strategies

- On a write miss, is the block loaded from memory into the cache?

**Write Allocate:**
- Block is loaded into cache on a write miss.
- Usually used with write back
- Otherwise, write-back requires read-modify-write to replace word within block

- But if you’ve gone to the trouble of reading the entire block, why not load it in cache?
Allocation Strategies (Cont.)

- On a write miss, is the block loaded from memory into the cache?

No-Write Allocate (Write Around):
- Block is not loaded into cache on a write miss
- Usually used with write through
  - Memory system directly handles word-level writes
Qualitative Cache Performance Model

**Miss Types**
- **Compulsory ("Cold Start") Misses**
  - First access to line not in cache
- **Capacity Misses**
  - Active portion of memory exceeds cache size
- **Conflict Misses**
  - Active portion of address space fits in cache, but too many lines map to same cache entry
  - Direct mapped and set associative placement only
- **Coherence Misses**
  - Block invalidated by multiprocessor cache coherence mechanism

**Hit Types**
- **Reuse hit**
  - Accessing same word that previously accessed
- **Line hit**
  - Accessing word spatially near previously accessed word
Major Cache Effects to Consider

• Total cache size
  - Try to keep heavily used data in highest level cache

• Block size (sometimes referred to “line size”)
  - Exploit spatial locality

Example Application

• Multiply \( n \times n \) matrices
• \( O(n^3) \) total operations
• Accesses
  - \( n \) reads per source element
  - \( n \) values summed per destination
    » But may be able to hold in register

```c
/* ijk */
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```
Matmult Performance (Alpha 21164)

Too big for L1 Cache
Too big for L2 Cache
Block Matrix Multiplication

Example $n=8$, $B = 4$:

\[
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
= \begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
\]

Key idea: Sub-blocks (i.e., $A_{ij}$) can be treated just like scalars.

\[
C_{11} = A_{11}B_{11} + A_{12}B_{21} \\
C_{12} = A_{11}B_{12} + A_{12}B_{22} \\
C_{21} = A_{21}B_{11} + A_{22}B_{21} \\
C_{22} = A_{21}B_{12} + A_{22}B_{22}
\]
Blocked Matrix Multiply (bijk)

for (jj=0; jj<n; jj+=bsize) {
    for (i=0; i<n; i++)
        for (j=jj; j < min(jj+bsize,n); j++)
            c[i][j] = 0.0;
    for (kk=0; kk<n; kk+=bsize) {
        for (i=0; i<n; i++) {
            for (j=jj; j < min(jj+bsize,n); j++) {
                sum = 0.0
                for (k=kk; k < min(kk+bsize,n); k++) {
                    sum += a[i][k] * b[k][j];
                }
                c[i][j] += sum;
            }
        }
    }
}
Blocked Matrix Multiply Analysis

- Innermost loop pair multiplies 1 \times bsize sliver of A times bsize \times bsize block of B and accumulates into 1 \times bsize sliver of C
- Loop over \( i \) steps through \( n \) row slivers of A & C, using same B

\[
\begin{align*}
\text{for } (i=0; i<n; i++) & \{ \\
& \quad \text{for } (j=jj; j < \min(jj+bsize,n); j++) \{ \\
& \quad \quad \text{sum} = 0.0 \\
& \quad \quad \text{for } (k=kk; k < \min(kk+bsize,n); k++) \{ \\
& \quad \quad \quad \text{sum} += a[i][k] \times b[k][j]; \\
& \quad \quad \} \\
& \quad \text{c}[i][j] += \text{sum}; \\
& \} 
\end{align*}
\]

Innermost Loop Pair

- Innermost loop pair multiplies 1 \times bsize sliver of A times bsize \times bsize block of B and accumulates into 1 \times bsize sliver of C
- Loop over \( i \) steps through \( n \) row slivers of A & C, using same B

Update successive elements of sliver

row sliver accessed bsize times

block reused \( n \) times in succession
Blocked matmult perf (Alpha 21164)
Why VM?: 1) DRAM a “Cache” for Disk

The full address space is quite large:

- 32-bit addresses: \(~4,000,000,000\) (4 billion) bytes
- 64-bit addresses: \(~16,000,000,000,000,000,000\) (16 quintillion) bytes

Disk storage is \(~30X\) cheaper than DRAM storage

- 8 GB of DRAM: \(~$12,000\)
- 8 GB of disk: \(~$200\)

To access large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk.
Levels in Memory Hierarchy

- **CPU registers**
- **Cache**: 32 KB / 4 MB, 128 MB
- **Memory**: 6 ns, 60 ns
- **Disk Memory**: 20 GB, 8 ms
- **Register**:
  - size: 200 B
  - speed: 3 ns
  - $/Mbyte: $100/MB, $1.50/MB
  - block size: 8 B

Larger, slower, cheaper
DRAM vs. SRAM as a “Cache”

DRAM vs. disk is more extreme than SRAM vs. DRAM

- access latencies:
  - DRAM is ~10X slower than SRAM
  - disk is ~100,000X slower than DRAM

- importance of exploiting spatial locality:
  - first byte is ~100,000X slower than successive bytes on disk
  - vs. ~4X improvement for page-mode vs. regular accesses to DRAM

- “cache” size:
  - main memory is ~100X larger than an SRAM cache

- addressing for disk is based on sector address, not memory address
If DRAM was to be organized similar to an SRAM cache, how would we set the following design parameters?

- Line size?
- Associativity?
- Replacement policy (if associative)?
- Write through or write back?

What would the impact of these choices be on:

- miss rate
- hit time
- miss latency
- tag overhead
Locating an Object in a “Cache”

1. Search for matching tag
   - SRAM cache

<table>
<thead>
<tr>
<th>Object Name</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>1</td>
</tr>
</tbody>
</table>

   = X?

2. Use indirection to look up actual object location
   - virtual memory

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>D</td>
</tr>
<tr>
<td>1:</td>
<td>X</td>
</tr>
<tr>
<td>N-1:</td>
<td>J</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>D: 0</td>
<td>243</td>
</tr>
<tr>
<td>J: N-1</td>
<td></td>
</tr>
<tr>
<td>X: 1</td>
<td>17</td>
</tr>
<tr>
<td>N-1:</td>
<td>105</td>
</tr>
</tbody>
</table>
A System with Physical Memory Only

Examples:
- most Cray machines, early PCs, nearly all embedded systems, etc.

CPU's load or store addresses used directly to access memory.
A System with Virtual Memory

Examples:
- workstations, servers, modern PCs, etc.

Address Translation: the hardware converts virtual addresses into physical addresses via an OS-managed lookup table (page table)
Page Faults (Similar to “Cache Misses”)

What if an object is on disk rather than in memory?
- Page table entry indicates that the virtual address is not in memory
- An OS trap handler is invoked, moving data from disk into memory
  - current process suspends, others can resume
  - OS has full control over placement, etc.
Servicing a Page Fault

**Processor Signals Controller**
- Read block of length P starting at disk address X and store starting at memory address Y

**Read Occurs**
- Direct Memory Access
- Under control of I/O controller

**I/O Controller Signals Completion**
- Interrupt processor
- Can resume suspended process

Diagram:
- (1) Initiate Block Read
- (2) DMA Transfer
- (3) Read Done
Why VM? 2) Memory Management

Multiple processes can reside in physical memory. How do we resolve address conflicts?

(Virtual) Memory Image for Alpha Process

0000 03FF 8000 0000
Reserved

0000 0001 2000 0000
Not yet allocated

$gp

0000 0000 0001 0000
Stack

Not yet allocated

0000 0000 0000 0000
Reserved

Text (Code)

Static Data

Dynamic Data

$sp

e.g., what if two different Alpha processes access their stacks at address 0x11fffffff80 at the same time?
Soln: Separate Virtual Addr. Spaces

- Virtual and physical address spaces divided into equal-sized blocks
  - “Pages” (both virtual and physical)
- Each process has its own virtual address space
  - operating system controls how virtual pages are assigned to physical memory

Virtual Addresses

<table>
<thead>
<tr>
<th>Virtual Pages</th>
<th>Physical Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 1</td>
<td>PP 2</td>
</tr>
<tr>
<td>VP 2</td>
<td>PP 7</td>
</tr>
<tr>
<td>VP 1</td>
<td>PP 10</td>
</tr>
<tr>
<td>VP 2</td>
<td></td>
</tr>
</tbody>
</table>

Process 1:

Process 2:

0 0
N-1 M-1

(Read-only library code)
Page table entry contains access rights information
- hardware enforces this protection (trap into OS if violation occurs)

**Page Tables**

<table>
<thead>
<tr>
<th>Process i:</th>
<th>Read?</th>
<th>Write?</th>
<th>Physical Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 4</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>No</td>
<td>XXXXXXXX</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process j:</th>
<th>Read?</th>
<th>Write?</th>
<th>Physical Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>No</td>
<td>XXXXXXXX</td>
</tr>
</tbody>
</table>

**Memory**

0:
1:
N-1:
VM Address Translation

\[ V = \{0, 1, \ldots, N-1\} \] virtual address space \( N > M \)

\[ P = \{0, 1, \ldots, M-1\} \] physical address space

\( \text{MAP: } V \rightarrow P \cup \{\emptyset\} \) address mapping function

\[ \text{MAP}(a) = a' \quad \text{if data at virtual address } a \text{ is present at physical address } a' \text{ in } P \]

\[ = \emptyset \quad \text{if data at virtual address } a \text{ is not present in } P \]

Processor

\( \emptyset \)

Addr Trans Mechanism

fault handler

Main Memory

Secondary memory

physical address

missing item fault

OS performs this transfer (only if miss)
VM Address Translation

Parameters
- $P = 2^p = \text{page size (bytes)}$. Typically 4KB–64KB
- $N = 2^n = \text{Virtual address limit}$
- $M = 2^m = \text{Physical address limit}$

Notice that the page offset bits don't change as a result of translation.
Page Tables

Virtual Page Number

Page Table
(physical page or disk address)

Valid

Physical Memory

Disk Storage

1
1
0
1
1
1
0
1
0
1
Address Translation via Page Table

- Virtual page number
- Page offset
- Virtual address
- Physical page number
- Page offset
- Physical address

VPN acts as a table index

If valid = 0, then page not in memory

Page table base register

Address
Page Table Operation

Translation
  • Separate (set of) page table(s) per process
  • VPN forms index into page table

Computing Physical Address
  • Page Table Entry (PTE) provides information about page
    - if (Valid bit = 1) then page in memory.
      » Use physical page number (PPN) to construct address
    - if (Valid bit = 0) then page in secondary memory
      » Page fault
      » Must load into main memory before continuing

Checking Protection
  • Access rights field indicate allowable access
    - e.g., read-only, read-write, execute-only
    - typically support multiple protection modes (e.g., kernel vs. user)
  • Protection violation fault if don’t have necessary permission
Integrating VM and Cache

Most Caches “Physically Addressed”
- Accessed by physical addresses
- Allows multiple processes to have blocks in cache at the same time
- Allows multiple processes to share pages
- Cache doesn’t need to be concerned with protection issues
  - Access rights checked as part of address translation

Perform Address Translation Before Cache Lookup
- But this could involve a memory access itself
- Of course, page table entries can also become cached
Speeding up Translation with a TLB

“Translation Lookaside Buffer” (TLB)
- Small, usually fully associative cache
- Maps virtual page numbers to physical page numbers
- Contains complete page table entries for small number of pages

Diagram:
- CPU
- TLB Lookup
- Translation
- Cache
- Main Memory
- VA to PA
- Hit/miss
- Data
- Miss/Hit
Address Translation with a TLB

- virtual address
- process ID
- virtual page number
- page offset
- virtual address
- validdirty
- tag
- physical page number
- physical address
- tag
- physical address
- data
- index
- byte offset
- data
- valid
- tag
- data
- TLB hit
- cache hit
- valid
- tag
- data
- TLB
- Cache
Alpha AXP 21064 TLB

- Page size: 8KB
- Hit time: 1 clock
- Miss penalty: 20 clocks
- TLB size: ITLB 8 PTEs, DTLB 32 PTEs
- Placement: Fully associative
TLB-Process Interactions

TLB Translates Virtual Addresses
  • But virtual address space changes each time have context switch

Could Flush TLB
  • Every time perform context switch
  • Refill for new process by series of TLB misses
  • ~100 clock cycles each

Could Include Process ID Tag with TLB Entry
  • Identifies which address space being accessed
  • OK even when sharing physical pages
Virtually-Indexed Cache

**Cache Index Determined from Virtual Address**
- Can begin cache and TLB index at same time

**Cache Physically Addressed**
- Cache tag indicates physical address
- Compare with TLB result to see if match
  - Only then is it considered a hit

**What extra info needs to be included in cache?**
**When Can Cache be virtually tagged?**
Generating Index from Virtual Address

Size cache so that index is determined by page offset
- Can increase associativity to allow larger cache
- E.g., early PowerPC’s had 32KB cache
  - 8-way associative, 4KB page size

Page Coloring
- Make sure lower k bits of VPN match those of PPN
- Page replacement becomes set associative
- Number of sets = $2^k$
Single level page table:
- 64-bit address space
- 4kb page size
- 512MB physical memory

Page table entries: $2^{64}/2^{12}$ entries!
Entry size: $2^{29}/2^{12} \rightarrow 17$ bits per entry + other
Page table size: $2^{52} \times 2^{2} = 2^{54}!!
Alpha Virtual Addresses

Page Size
- 8KB (and some multiples)

Page Tables
- Each table fits in single page
- Page Table Entry 8 bytes
  - 4 bytes: physical page number
  - Other bytes: for valid bit, access information, etc.
- 8K page can have 1024 PTEs

Alpha Virtual Address
- Based on 3-level paging structure

<table>
<thead>
<tr>
<th>level 1</th>
<th>level 2</th>
<th>level 3</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>13</td>
</tr>
</tbody>
</table>

- Each level indexes into page table
- Allows 43-bit virtual address when have 8KB page size
Alpha Page Table Structure

Tree Structure
- Node degree $\leq 1024$
- Depth = 3

Nice Features
- No need to enforce contiguous page layout
- Dynamically grow tree as memory needs increase
Mapping an Alpha 21064 Virtual Address

- Virtual address
  - seg0/seg1: 000 ... 0 or 111 ... 1
  - Level1
  - Level2
  - Level3
  - Page offset

- Physical address
  - Physical page-frame number
  - Page offset

- Main memory

- L1 page table
- L2 page table
- L3 page table

- PTE size: 8 Bytes
- PT size: 1024 PTEs

- 10 bits
- 13 bits
- 21 bits
## Virtual Address Ranges

<table>
<thead>
<tr>
<th>Binary Address Pattern</th>
<th>Segment</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>1...1 11 xxxx...xxx</td>
<td>seg1</td>
<td>Kernel accessible virtual addresses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Information maintained by OS but not to be accessed by user</td>
</tr>
<tr>
<td>1...1 10 xxxx...xxx</td>
<td>kseg</td>
<td>Kernel accessible physical addresses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- No address translation performed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Used by OS to indicate physical addresses</td>
</tr>
<tr>
<td>0...0 0x xxxx...xxx</td>
<td>seg0</td>
<td>User accessible virtual addresses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Only part accessible by user program</td>
</tr>
</tbody>
</table>

### Address Patterns

- **Must have high order bits all 0’s or all 1’s**
  - Currently 64-43 = 21 wasted bits in each virtual address
- **Prevents programmers from sticking in extra information**
  - Could lead to problems when want to expand virtual address space in future
## Alpha Seg0 Memory Layout

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3FF FFFF FFFF</td>
<td>Reserved (shared libraries)</td>
</tr>
<tr>
<td>3FF 8000 0000</td>
<td>Not yet allocated</td>
</tr>
<tr>
<td></td>
<td>Dynamic Data</td>
</tr>
<tr>
<td>001 4000 0000</td>
<td>Static Data</td>
</tr>
<tr>
<td></td>
<td>Not used</td>
</tr>
<tr>
<td>001 2000 0000</td>
<td>Text (Code)</td>
</tr>
<tr>
<td></td>
<td>Stack</td>
</tr>
<tr>
<td></td>
<td>Not yet allocated</td>
</tr>
<tr>
<td>000 0001 0000</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

### Regions

- **Data**
  - Static space for global variables
    - Allocation determined at compile time
    - Access via $gp
  - Dynamic space for runtime allocation
    - E.g., using `malloc`
- **Text**
  - Stores machine code for program
- **Stack**
  - Implements runtime stack
  - Access via $sp
- **Reserved**
  - Used by operating system
    - Shared libraries, process info, etc.
Alpha Seg0 Memory Allocation

Address Range
- User code can access memory locations in range
  0x0000000000010000 to 0x000003FFFFFF
- Nearly $2^{42} \approx 4.3980465 \times 10^{12}$ byte range
- In practice, programs access far fewer

Dynamic Memory Allocation
- Virtual memory system only allocates blocks of memory as needed
- As stack reaches lower addresses, add to lower allocation
- As break moves toward higher addresses, add to upper allocation
  - Due to calls to malloc, calloc, etc.
## Minimal Page Table Configuration

### User-Accessible Pages

- **VP4: Shared Library**
  - Read only to prevent undesirable interprocess interactions
  - Near top of Seg0 address space

- **VP3: Data**
  - Both static & dynamic
  - Grows upward from virtual address 0x140000000

- **VP2: Text**
  - Read only to prevent corrupting code

- **VP1: Stack**
  - Grows downward from virtual address 0x120000000
Partitioning Addresses

- **Address**: 0x001 2000 0000
  - Level 1: 0
  - Level 2: 576
  - Level 3: 0

- **Address**: 0x001 4000 0000
  - Level 1: 0
  - Level 2: 640
  - Level 3: 0

- **Address**: 0x3FF 8000 0000
  - Level 1: 511
  - Level 2: 768
  - Level 3: 0
### Increasing Heap Allocation

**Without More Page Tables**
- Could allocate 1023 additional pages
- Would give ~8MB heap space

**Adding Page Tables**
- Must add new page table with each additional 8MB increment

**Maximum Allocation**
- Our Alphas limit user to 1GB data segment
- Limit stack to 32MB

![Diagram showing heap allocation and page tables]

```
0000 0001 4000 0000
0000 0001 4000 007F
0000 0001 4000 1FFF
0000 0001 4000 2000
0000 0001 4000 3FFF
```

```
0000 0001 4000 0000
0000 0001 4000 1023
```

```
0000 0001 407F FFFF
0000 0001 4000 2000
```

```
0000 0001 4000 0000
```

```
0000 0001 4000 007F
```
Expanding Alpha Address Space

Increase Page Size

- Increasing page size 2X increases virtual address space 16X
  - 1 bit page offset, 1 bit for each level index

<table>
<thead>
<tr>
<th>level 1</th>
<th>level 2</th>
<th>level 3</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>10+k</td>
<td>10+k</td>
<td>10+k</td>
<td>13+k</td>
</tr>
</tbody>
</table>

Physical Memory Limits

- Cannot be larger than kseg
  VA bits -2 ≥ PA bits
- Cannot be larger than 32 + page offset bits
  - Since PTE only has 32 bits for PPN

Configurations

- Page Size  8K  16K  32K  64K
- VA Size    43  47  51  55
- PA Size    41  45  47  48
Page Size Trade-offs

- As page size increases?

- As Page size decreases?
VM Theme

Programmer's View
- Large “flat” address space
  - Can allocate large blocks of contiguous addresses
- Process “owns” machine
  - Has private address space
  - Unaffected by behavior of other processes

System View
- User virtual address space created by mapping to set of pages
  - Need not be contiguous
  - Allocated dynamically
  - Enforce protection during address translation
- OS manages many processes simultaneously
  - Continually switching among processes
  - Especially when one must wait for resource
    » E.g., disk I/O to handle page fault
The Memory Hierarchy

- Exploits locality to give appearance of a large, fast flat address space owned by a process
  - cache(s)
  - VM
  - TLB
- Hierarchical to exploit qualities of each technology while reducing cost of overall system
- Interaction between cache design and VM design
- What should be automatic and what “manual”?  
  - Registers?  Cache placement?  TLB management?